**Pipelinea**

**4.5 Uma visão geral do Pipeline**

* O que é pipeline? É uma técnica de implementação onde várias instruções são sobrepostas na execução; assim que uma instrução termina de executar o seu primeiro estágio e parte para o segundo, a próxima instrução já ocupa o primeiro estágio; hoje em dia, é uma técnica praticamente universal.
* Objetivo: iniciar a execução de uma instrução antes da anterior ter terminado, pois se várias instruções forem processadas ao mesmo tempo, com cada parte do hardware atuando em uma instrução distinta, teríamos uma melhor utilização do hardware.
* Paralelismo de execução de instruções: aumenta o número de instruções sendo executadas ao mesmo tempo e também aumenta o ritmo de instruções iniciadas e finalizadas.
* Desde que existam recursos separados para cada estágio de um conjunto de instruções, ele pode ser executado usando a técnica de pipeline.
* Determinados conjuntos de instruções podem facilitar ou dificultar a execução de um pipeline.
* A principal razão do pipeline ser uma técnica muito mais rápida é que os estágios estão ocorrendo em paralelo. Isso faz com que o throughput (taxa de execução) aumente.
* Usar pipeline não diminui o tempo necessário para completar uma única instrução, porém, na existência de várias instruções, o aumento no throughput diminui o tempo total para completar todo o trabalho; aumentando o desempenho.
* Se todos os estágios duram aproximadamente o mesmo tempo, o speed-up do pipeline é igual ao número de estágios presentes no pipeline.
* *Os 5 estágios do pipeline*:
  + **1.** Busca de instrução na memória.
  + **2.** Leitura de registradores e decodificação da instrução.
  + **3.** Execução da operação ou cálculo de endereço.
  + **4.** Leitura da memória de dados.
  + **5.** Escrita de um dado em registrador.
* O tempo de cada estágio do pipeline é definido pelo recurso mais demorado, normalmente uma operação da ULA ou um acesso à memória.
* Observação: assumimos que a escrita no registrador ocorre na primeira metade do estágio e que a lida ocorre na segunda metade.

**Tempo entre instruções no pipeline:**

* O RISC-V foi arquitetado com o objetivo do uso de pipeline:
  + No RISC-V, todas as instruções tem o mesmo tamanho: essa restrição torna muito mais fácil a busca de instruções no primeiro estágio e a decodificação delas no segundo estágio.
  + O RISC-V tem apenas poucos formatos de instruções: elas possuem os registradores fonte e destino localizados sempre nos mesmos campos em cada instrução.
  + No RISC-V, apenas loads e stores fazem acesso à memória: essa restrição faz com que possamos usar o estágio de execução de operação para calcular o endereço de memória e já no estágio seguinte acessá-la.

**Hazards:**

* Situações no pipeline em que a próxima instrução não pode ser executada no clock seguinte.
* Structural Hazard/Conflitos Estruturais: quando uma instrução não pode ser executada no ciclo adequado, pois o hardware não suporta essa determinada combinação de instruções que estão programadas para executar.
  + Como o RISC-V foi estruturado para o pipeline, é mais difícil a ocorrência de structural hazards.
  + Se, por exemplo, existisse apenas uma memória no caminho de dados, o estágio de busca de instrução e o estágio de acesso à memória para escrita/leitura não poderiam ser executados ao mesmo tempo.
  + Por isso, o pipeline do RISC-V exige duas memórias (uma de instruções e uma de dados).
* Data Hazard: quando uma instrução não pode ser executada no ciclo adequado, pois um dado necessário para executar a instrução ainda não está disponível.
  + *add x19, x0, x1*
  + *sub x2, x19, x3*
  + O dado só é escrito em x19 no último estágio, o que significa que a leitura dos registradores da segunda instrução não conteria o valor correto de x19.

**Forwarding or Bypassing:**

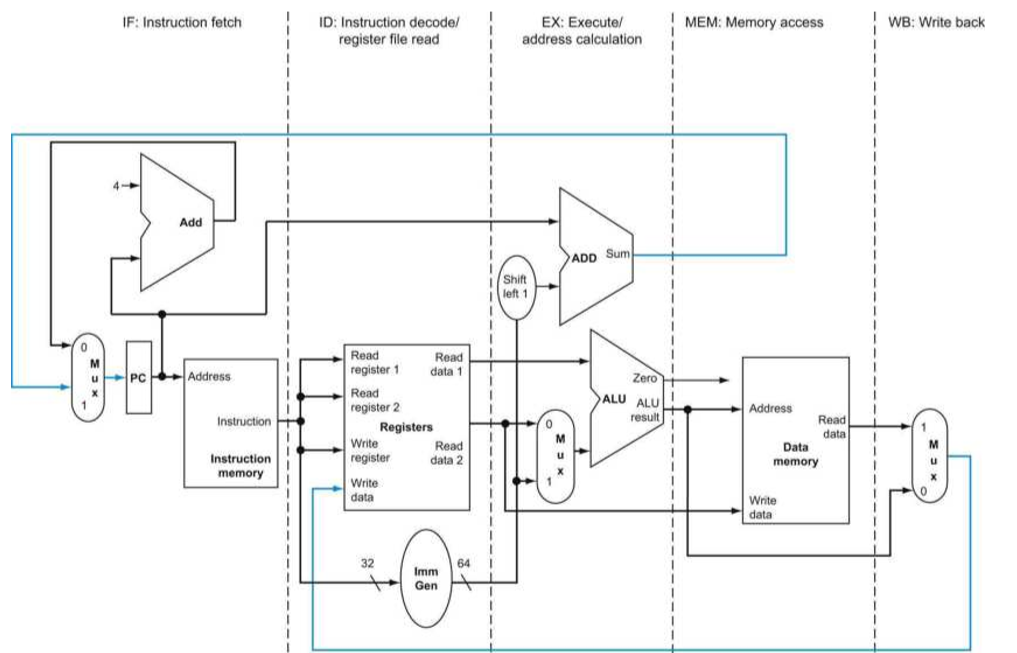
* Usando como exemplo a situação anterior, assim que a ALU calcular a soma, poderíamos enviar como input o resultado para a subtração.
* Conceito: adição de hardware extra para ter acesso à uma informação mais rapidamente; captar dos buffers/registradores internos um dado necessário ao invés de esperar por ele ficar pronto (armazenado em memória ou registrador).
* Forwarding só pode ser usado se o estágio destino é executado em um tempo mais a frente do que o estágio fonte.
* Tendo como exemplo um load de x1 seguido de um sub usando x1: o dado em x1 só estaria disponível após o quarto estágio, que é muito tarde para o input do terceiro estágio da sub.
* Ou seja, mesmo usando forwarding, ainda sim as vezes se torna necessário outra estratégia.

**Hazards:**

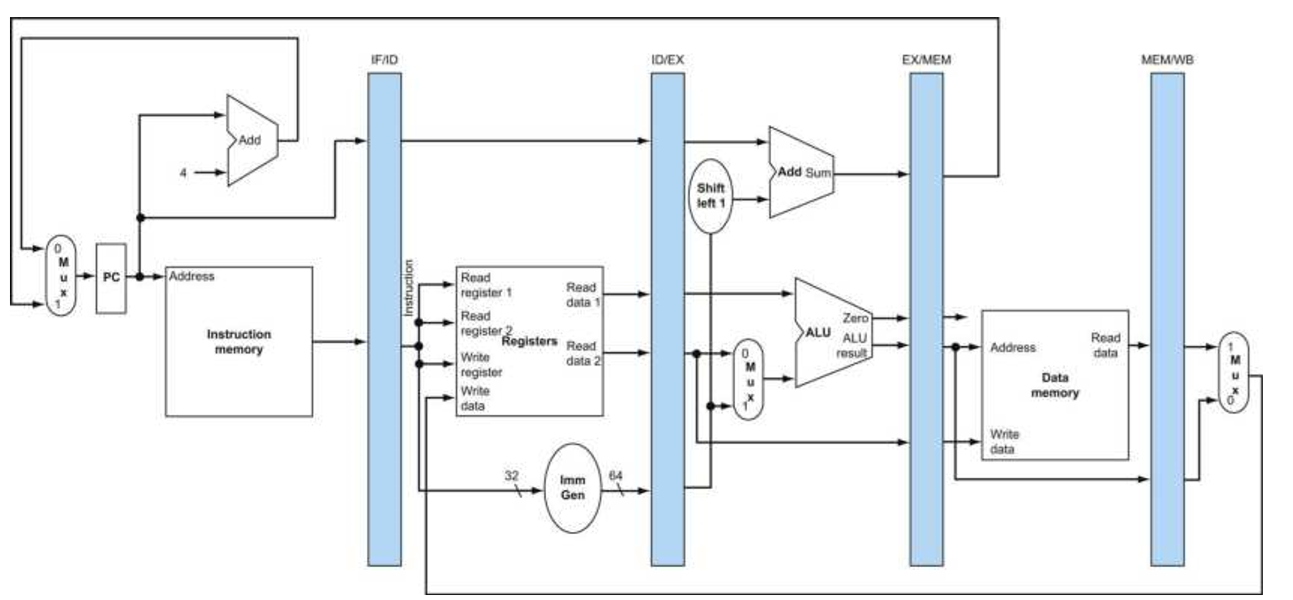
* Load-use Data Hazard: um tipo específico de data hazard; quando um dado está sendo carregado pelo load e ele ainda não está disponível, porém já é necessário para outra instrução.
* Pipeline Stall/Bubble: usado para “atrasar” as instruções, com o objetivo de resolver um hazard.
* Hazards podem ser resolvidos de duas formas: detecção por hardware e uso de bolhas ou uso de um software que reorganiza o código para evitar que load-use data hazards ocorram.
* Atenção: quando uma instrução R-type deve ser executada depois de um load, são necessários tanto o forwarding quanto o uso de bolhas.
* Control Hazard/Branch Hazard: necessidade de tomar uma decisão baseada nos resultados de uma instrução enquanto outras instruções estão executando; quando a instrução correta não pode ser executada no momento correto porque a instrução que foi buscada não é a que é necessária, ou seja, o fluxo das instruções não é o que o pipeline esperava.
  + *BEQ:* devemos buscar a instrução que segue o branch no próximo clock, porém o pipeline não tem como prever qual deve ser a próxima instrução (se o desvio ocorrerá ou não); uma possível solução é utilizar uma bolha logo após a busca do branch, esperando até que o pipeline determine o resultado do branch e saiba de certeza qual a próxima instrução que deve ser executada.
  + Mesmo se existir hardware extra que torne possível que a avaliação do desvio, o cálculo do endereço e a atualização do PC ocorram no segundo estágio, ainda seria necessário o uso de uma bolha, pois é necessário aguardar até que o resultado da condição do desvio seja determinado antes de buscar a próxima instrução.
  + Se não é possível realizar o branch no segundo estágio, o atraso na pipeline será muito maior para os desvios condicionais (três bolhas).
  + Existe um outro método para solucionar esse tipo de hazard, baseado em previsões. Caso a previsão esteja certa, não ocorre atraso no pipeline. Porém, se estava errada, é necessário refazer a instrução. Nesse caso, a instrução correta.
  + Uma estratégia é supor que o desvio nunca será executado. Apenas quando houver desvio é que ocorrerá algum tipo de atraso no pipeline.
  + Uma versão mais sofisticada é a branch prediction: alguns desvios seriam previstos como executados e alguns como não executados. Por exemplo, desvios que voltam para uma parte anterior do código normalmente são previstos para que ocorram (devido a loops). É o método que assume um certo resultado para o desvio e já executa a próxima instrução de acordo com essa previsão, ao invés de esperar pelo resultado da instrução de desvio.
  + Algumas estratégias de previsão se baseiam no estereótipo dos desvios; as previsões são fixas e feitas previamente; baseadas no comportamento típico dos desvios; prever que desvios para partes anteriores do código ocorrerão e prever que desvios para partes posteriores do código não ocorrerão.
  + Previsões dinâmicas fazem as previsões dependendo no comportamento de cada desvio e podem mudar seus palpites durante a execução do programa; ajustando a próxima previsão dependendo do sucesso da corrente; se mantém um histórico para cada desvio como executados ou não executados e esse histórico é usado para prever o próximo desvio.
  + Caso a previsão seja incorreta, o controle do pipeline deve garantir que a instrução que foi prevista erroneamente não afete o código e deve recomeçar o pipeline agora a partir da instrução correta.
  + *MIPS*: delayed decision; o software coloca outra instrução que não é afetada pelo desvio imediatamente seguindo-o, e caso o desvio ocorra, o software muda o endereço da instrução que deve ser executada após essa instrução de garantia.

**4.6 Caminho de Dados e Unidade de Controle no Pipeline**

* A divisão de uma instrução em 5 estágios do pipeline significa que até 5 instruções podem estar sendo executadas durante qualquer ciclo de clock.
* Por isso separamos o caminho de dados em 5 pedaços, cada um nomeado e responsável por um dos estágios:
  + **1.** IF: Instruction fetch
  + **2.** ID: Instruction decode and register file read
  + **3.** EX: Execution or address calculation
  + **4.** MEM: Data memory access
  + **5.** WB: Write back



* Esses cinco componentes correspondem à maneira que o caminho de dados é desenhado. Instruções e dados normalmente se movem da esquerda para direita por esses cinco estágios enquanto completam a execução.
* Atenção: dados que caminham da direita para a esquerda podem causar hazards.
* Porém, existem duas exceções para essa regra:
  + O estágio WB (write back) para realizar a escrita do resultado no registrador.
  + A seleção do próximo valor do PC, escolhendo entre o PC+4 ou o endereço de desvio.
* Usamos registradores para manter os dados com o objetivo de que cada porção de um único datapath possa estar executando uma instrução diferente, facilitando o compartilhamento de um único datapath.



* Os registradores são nomeados com os estágios que ele separa.
* Em caso de ocorrência de exceção, os valores desses registradores normalmente não são salvos.

**Load:**

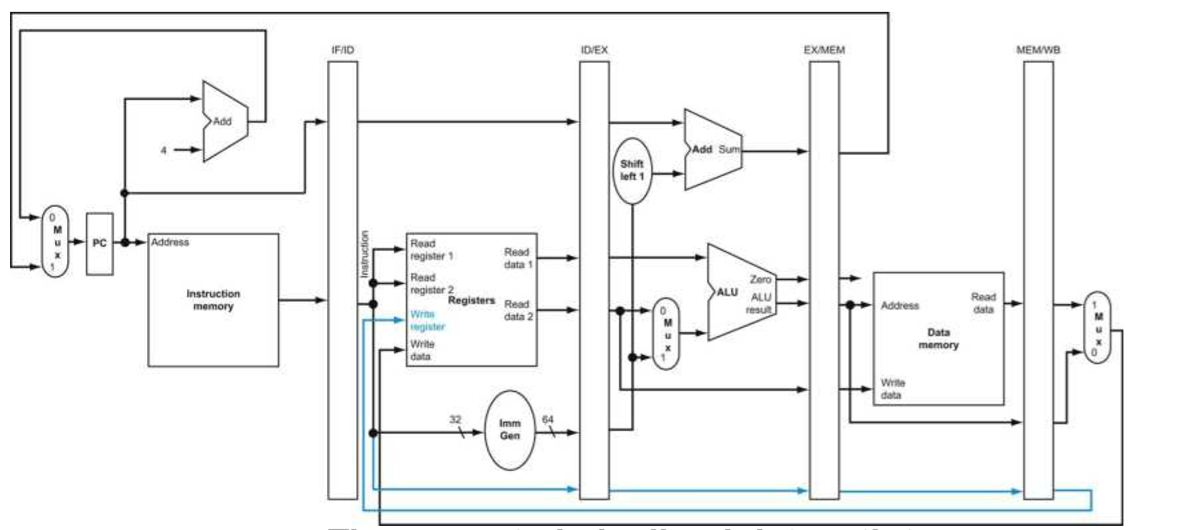
* **1. Instruction fetch:** a instrução é lida da memória usando o endereço armazenado no PC e é armazenada no registrador IF/ID; o valor do PC é incrementado em 4 e escrito (da direita para esquerda) no PC; esse novo valor do PC também é armazenado no registrador IF/ID caso seja posteriormente necessário para a instrução, como em um beq.
* **2. Instruction decode and register file read:** a instrução é lida do registrador IF/ID e é usada para cálculo do imediato e leitura dos registradores; esses 3 novos valores são armazenados no registrador ID/EX junto também com o valor do PC; além disso, o número do registrador de destino também é escrito no registrador ID/EX; mais um dado a ser passado no registrador ID/EX é o funct3 da instrução (pois caso fosse uma instrução de add/sub/addi/and, a unidade de controle necessitaria dessa informação para determinar qual a operação da ULA).
* **3. Execute or address calculation:** os conteúdos dos registradores e o imediato são lido do registrador ID/EX somados pela ULA para calcular o endereço de memória; essa soma é armazenada no registrador EX/MEM; além disso, é necessário guardar também no registrador EX/MEM o número do registrador destino, que está armazenado no registrador ID/EX.
* **4. Memory access:** o endereço de memória é lido do registrador EX/MEM e usado para ler da memória de dados; o dado lido é armazenado no registrador MEM/WB; além disso, armazenamos também o número do registrador destino no registrador MEM/WB.
* **5. Write back:** o dado lido da memória no estágio anterior é lido do registrador MEM/WB e escrito no registrador destino (que também tem seu número lido do registrador MEM/WB) (da direita para esquerda).

*Conclusão:* qualquer informação necessária em um estágio posterior deve ser passada entre os estágios por meio de registradores.

**Store:**

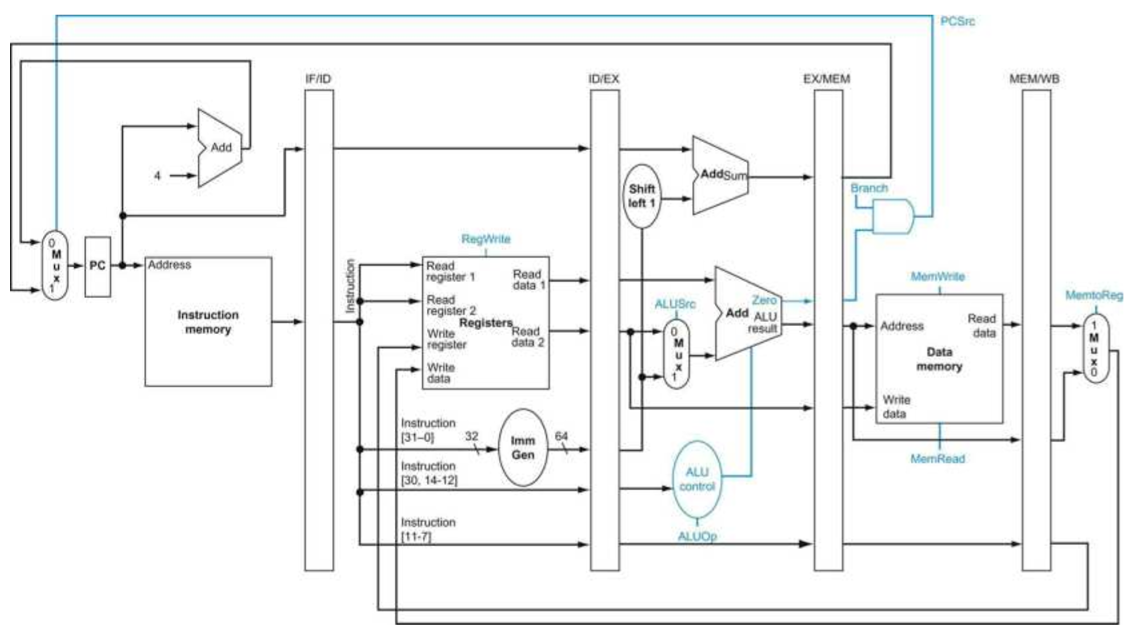
* **1. Instruction fetch:** a instrução é lida da memória a partir do endereço no PC e é guardada no registrador IF/ID; como esse estágio ocorre antes da identificação da instrução, é igual o primeiro estágio do load.
* **2. Instruction decode and register file read:** a instrução é lida do registrador IF/ID para que seja executada a leitura dos registradores e o cálculo do imediato; esses três dados são armazenados no registrador ID/EX; esse estágio também é similar ao segundo estágio do load; para todas as instruções funciona dessa maneira, pois ainda é cedo para determinar qual o tipo de instrução.
* **3. Execute or address calculation:** os conteúdos dos registradores e o imediato são lidos do registrador ID/EX somados pela ULA para calcular o endereço de memória; essa soma é armazenada no registrador EX/MEM; além disso, como precisamos do dado que foi lido do registrador no estágio 2 para ser escrito na memória, ele também é armazenado no registrador EX/MEM.
* **4. Memory access:** o dado a ser escrito na memória e o endereço são lidos do registrador EX/MEM; após isso, o dado é escrito na memória.
* **5. Write back:** nesse estágio, nada acontece para a instrução do store; ou seja, uma instrução passa por um estágio mesmo que não realize nenhuma tarefa nele, pois as próximas instruções já estão executando em outros estágios.
* Load e store exemplificam um ponto muito importante: cada componente lógico do pipeline só pode ser usado no seu estágio específico; caso o contrário, ocorreria structural hazards; dessa forma, cada componente e o seus controles podem ser associados a um único estágio do pipeline.

**Pipeline correto para o load/store:**

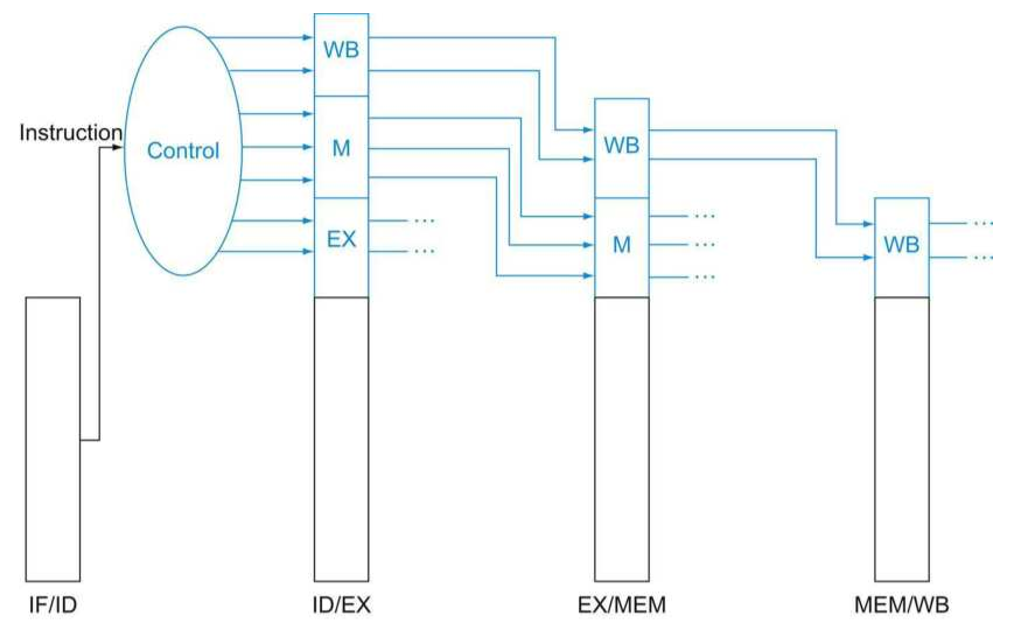


* Existem duas formas de representar o pipeline:
  + *multiple-clock-cycle pipeline diagrams*: mostram todos os estágios de várias instruções; instruções são executadas de cima para baixo e os estágios ocorrem da esquerda para a direita.
  + *single-clock-cycle pipeline diagrams*: mostra apenas um estágio de uma instrução; mostra com detalhes o uso do datapath em cada estágio, em cada ciclo.

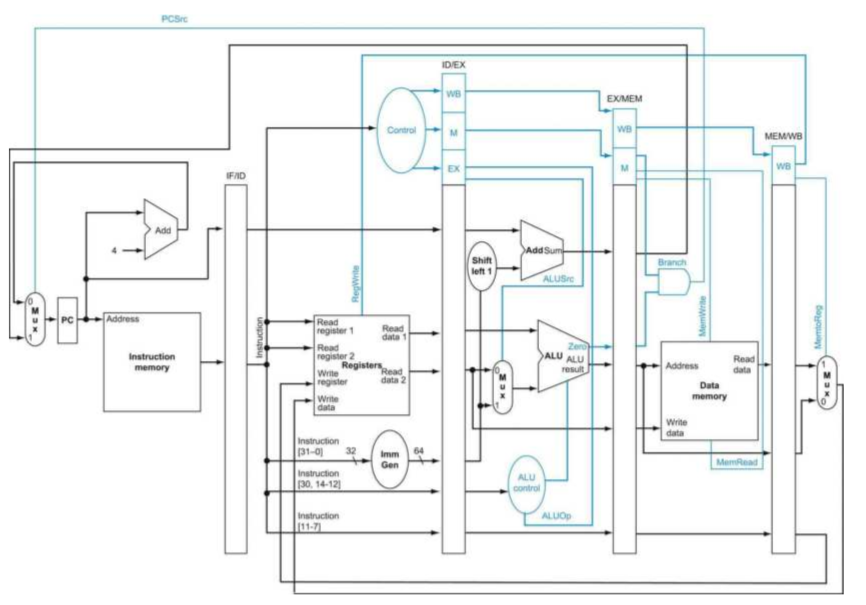
**Pipelined Control:**



* O PC é escrito a cada ciclo de clock, logo não precisa de um sinal de escrita separado, como o PCWrite (usado na implementação multiciclo).
* Usando o mesmo raciocínio, os registradores IF/ID, ID/EX, EX/MEM E MEM/WB também não precisam, são escritos a cada ciclo de clock também.
* Podemos dividir os sinais de controle em cinco grupos:
  + ***1.*** *Instruction fetch:* sinal de controle para leitura da memória de instruções e sinal de controle para escrever o PC são sempre necessários, para todos os ciclos de clock, logo, nesse estágio, não é necessária a unidade de controle.
  + ***2.*** *Instruction decode/register file read:* como os dois registradores para serem lidos do banco de registradores estão sempre na mesma posição da instrução no RISC-V, não há nada para controlar nesse estágio também.
  + ***3.*** *Execution/address calculation:* é necessário setar o sinal ALUOp (que determina a operação da ULA) e o sinal ALUSrc (que determina se a entrada da ULA será o registrador lido(0) ou o imediato calculado(1)).
  + ***4.*** *Memory Access:* três sinais pertencem a esse estágio; o Branch (1) que corresponde a instrução do BEQ, o MemRead (1) que corresponde a instrução de Load e o MemWrite (1) que corresponde a instrução de Store; o PCSrc será setado (1, se fosse 0 passaria o PC+4) para deixar passar o endereço de desvio caso seja uma instrução de BEQ e o sinal Zero da ULA esteja ativado.
  + ***5.*** *Write Back:* o sinal de controle MemtoReg (que envia o resultado da ULA(0) ou o que foi lido da memória para ser armazenado no registrador(1)) e o sinal RegWrite (habilitando a escrita no registrador destino(1)).
* Como os sinais de controle começam a ser utilizados no estágio EX, todas as informações de controle podem ser criadas durante o ID.
* A maneira mais fácil de passar esses sinais de controle para todos os estágios é estendendo os registradores do pipeline para que eles armazenem os sinais de controle.

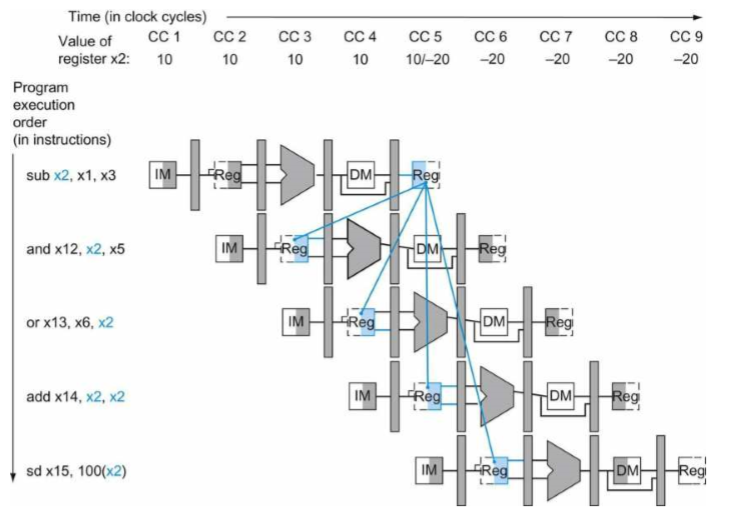


* São 7 sinais de controle, os 7 são passados para o estágio EX.
* *O estágio EX usa 2 dos sinais* e passa os outros 5 para o estágio MEM.
* *O estágio MEM usa 3 dos sinais* e passa os outros 2 para o estágio WB.
* *O estágio WB usa os 2 sinais restantes*.

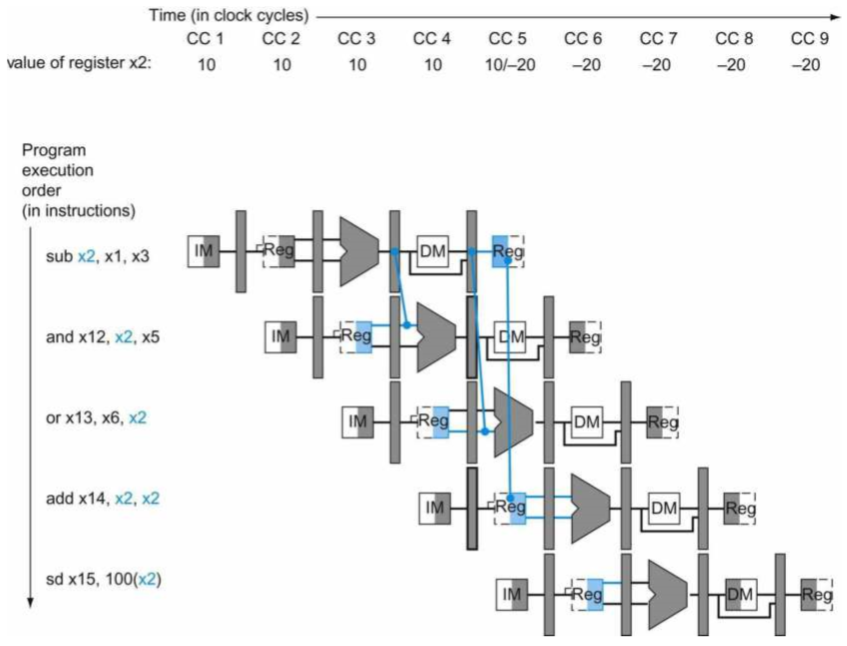


**4.7 Data Hazards: Fowarding vs Stalling (Bolhas)**

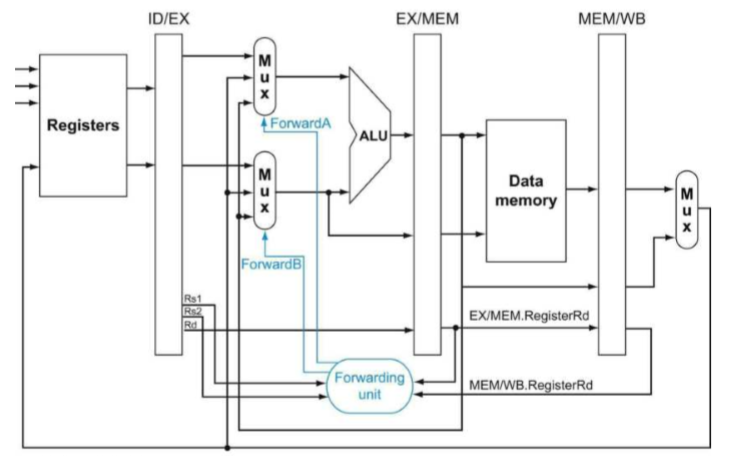
* Observe a sequência abaixo:
  + sub x2, x1, x3
  + and x12, x2, x5
  + or x13, x6, x2
  + add x14, x2, x2
  + sd x15, 100(x2)
* As últimas 4 instruções necessitam do valor de x2 determinado pela primeira instrução. Isso exemplifica um data hazard.
* O valor correto de x2 só está disponível após o último estágio da primeira instrução. Qualquer leitura de x2 que ocorra por outra instrução em um estágio anterior a essa ocorrência concederia o valor antigo de x2.
* Observação: O que ocorre quando um registrador é lido e escrito no mesmo clock? A escrita é realizada na primeira metade do ciclo e a leitura é realizada na segunda metade do ciclo, logo não existe data hazard nesse caso específico.



* Usando esse estilo de representação, data hazards ficam claramente evidenciados quando uma linha de dependência aparenta “voltar no tempo”.
* As instruções que apresentam problema são a segunda e a terceira.
* O dado correto de x2 já foi calculado no terceiro ciclo de clock, no estágio EX da primeira instrução. Ele é necessário no estágio EX das instruções and e or, que correspondem ao quarto e quinto ciclo de clock. Logo, basta usar o forwarding ao invés de bolhas para entregar o dado correto mais cedo à essas instruções, resolvendo o conflito.
* Forwarding para uma instrução no estágio EX: significa que quando uma instrução tenta usar o dado de um registrador que ainda vai ser escrito por uma instrução anterior, a entrada da ULA desta instrução deve ser a saída da ULA da instrução anterior.
* Hazard 1a:
  + EX/MEM.RegisterRD = ID/EX.RegisterRS1 = x2
* Hazard 1b:
  + EX/MEM.RegisterRd = ID/EX.RegisterRs2
* Hazard 2a:
  + MEM/WB.RegisterRd = ID/EX.RegisterRs1
* Hazard 2b:
  + MEM/WB.RegisterRD = ID/EX.RegisterRS2 = x2
* Como algumas instruções não escrevem em registradores, essa estratégia não é tão precisa, pois pode acabar executando o forwarding quando na verdade não deveria.
* *Uma solução:* checar se RegWrite é igual a 1 (nos estágios EX e MEM), caso seja, o forwarding provavelmente será necessário.
* Importante lembrar que no RISC-V o registrador x0 é constantemente igual a zero. Logo, se uma instrução tentar escrever em x0, devemos evitar o forwarding, pois provocaria um erro.



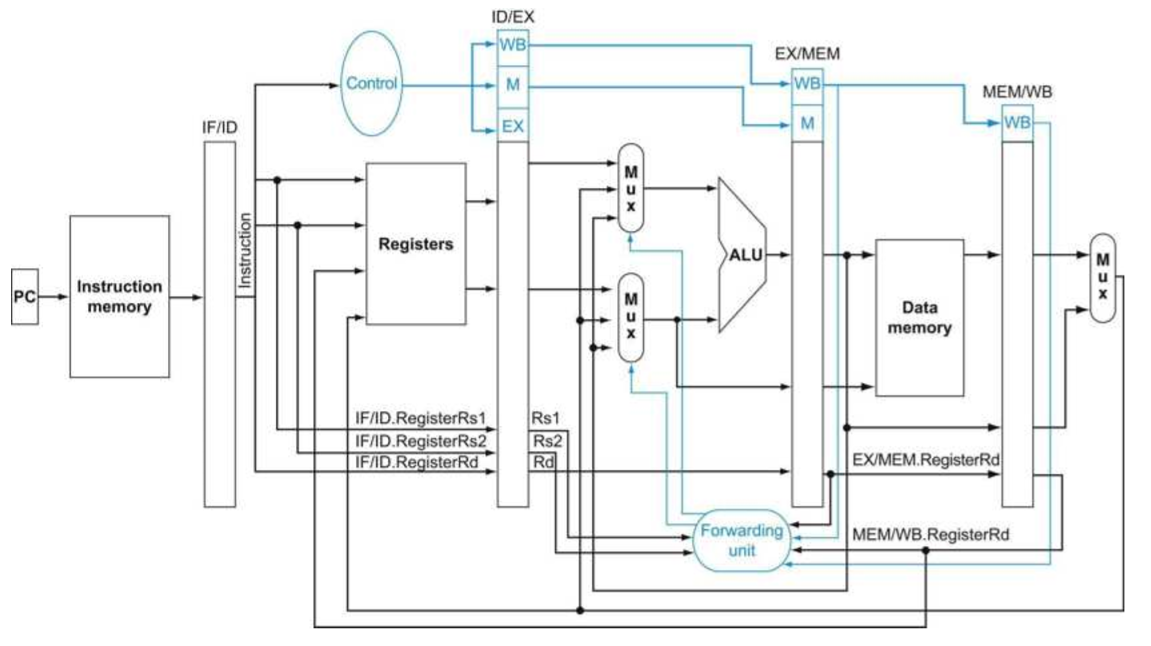
* Se pudermos carregar a entrada da ULA a partir de qualquer registrador do pipeline, poderemos sempre carregar o dado correto.
* Para isso, adicionamos mux’s para cada entrada da ULA.



* A unidade de forwarding fica localizada no estágio EX, pois a ULA e seus mux’s se encontram nesse estágio.
* Logo, devemos passar os registradores rs1 e rs2 para esta unidade para ela determinar se é necessário ou não o forwarding.
* Uma sequência de instruções que leem e escrevem no mesmo registrador:
  + add x1, x1, x2
  + add x1, x1, x3
  + add x1, x1, x4
* Nesse caso, o resultado deve ser forwarded a partir do estágio MEM, pois o resultado mais recente é o que se encontra nesse estágio.

**Caminho de dados que suporta data hazards:**

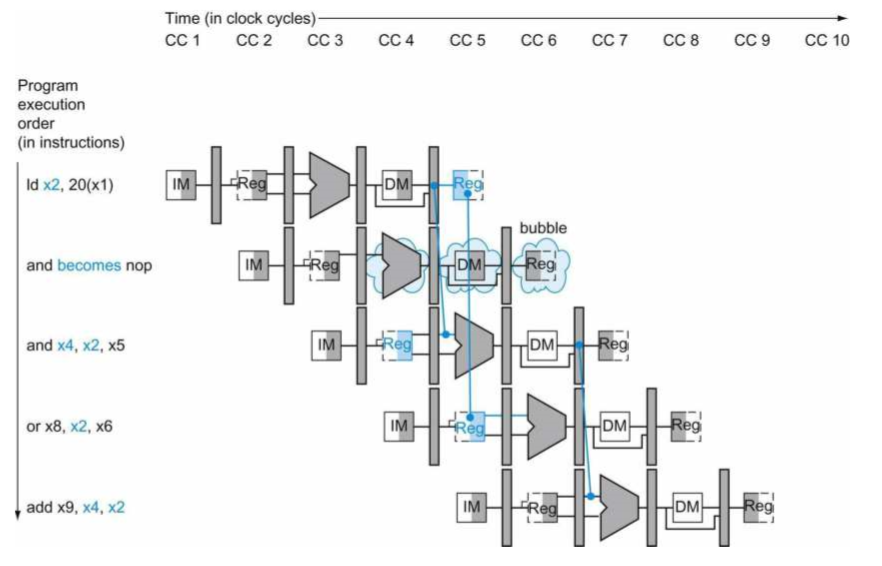
* As adições são os mux’s nas entradas da ULA. No circuito abaixo, alguns detalhes foram deixados de fora, como o hardware da instrução branch e a unidade de extensão de sinal.



* Forwarding também ajuda com hazards para instruções de store que estão dependendo de outras instruções.
* Porém para loads imediatamente seguidos por stores é necessário adicionar hardware extra. É possível evitar stall/bolha, pois o dado está no registrador MEM/WB da instrução load no momento do seu uso no estágio MEM do store.
* Seria necessário adicionar forwarding na memória de dados. Basta adicionar um mux na sua entrada controlado pela unidade de forwarding.
* Como a unidade de extensão de sinal não está no datapath e ela é necessária para loads e stores, precisamos arrumar uma maneira de fazer essa informação chegar na ULA quando necessária. Basta adicionar um outro MUX que possui como entradas o resultado do mux controlado pelo ForwardB e a extensão de sinal; esse mux que é controlado pelo sinal de controle ALUSrc.

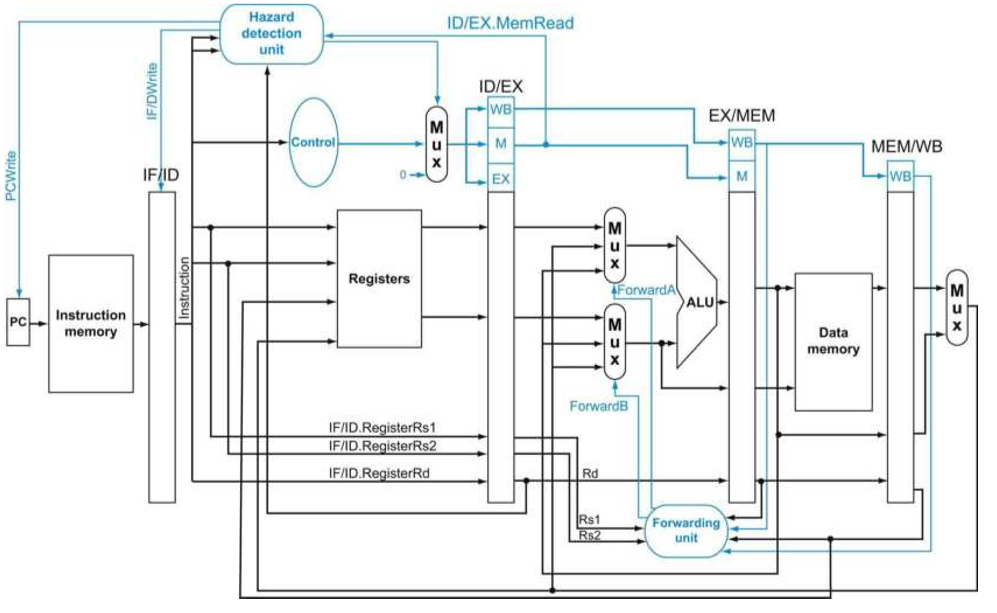
**Data Hazards e Bolhas:**

* Um caso que forwarding apenas não consegue solucionar é quando uma instrução tenta ler um registrador quando a instrução anterior é um load desse mesmo registrador.
* O dado ainda está sendo lido da memória no quarto ciclo de clock enquanto a ULA está realizando a operação da instrução seguinte. Algo deve atrasar o pipeline nesses tipos de casos.
* Além de uma unidade de forwarding, é necessária uma unidade de detecção de conflitos (hazard detection unit). Ela opera durante o estágio ID para poder inserir uma bolha entre o load e a instrução que depende dele.
* A unidade de detecção de conflitos checa se a instrução é um load, após isso checa se o registrador destino do load é igual a algum dos registradores fontes da instrução seguinte. Se tudo isso for verdade, a instrução é atrasada em um ciclo de clock. Após esse ciclo de clock, a unidade de fowarding adianta a informação lida da memória para a ULA da instrução seguinte resolvendo o problema.
* Caso não existisse o forwarding, seriam necessários dois ciclos de atraso, *duas bolhas*.
* Se a instrução no estágio ID é atrasada, a instrução no estágio IF também deve ser. Basicamente devemos fazer com que o valor do PC e do registrador IF/ID não mudem durante esse atraso.
* A parte do caminho de dados em que as instruções atrasadas estão contidas executam instruções que não possuem nenhum efeito:
  + Nops: uma instrução que não executa operação nenhuma; agem como bolhas.
* O que é realmente é feito: a desativação (0) dos 7 sinais da unidade de controle; mudamos os campos EX, MEM e WB do registrador ID/EX para zero; esses valores são passados para frente a cada ciclo de clock, criando a instrução nop.



* A unidade de controle de conflitos controla a escrita do PC e do registrador IF/ID, além disso controla um mux que escolhe entre o valor verdadeiro dos sinais de controle e zeros.
* Na realidade, apenas RegWrite e MemWrite devem ser setados como zero, o restante dos sinais podem ser don’t cares.

**Caminho de dados completo com Unidade de Detecção de Conflitos:**

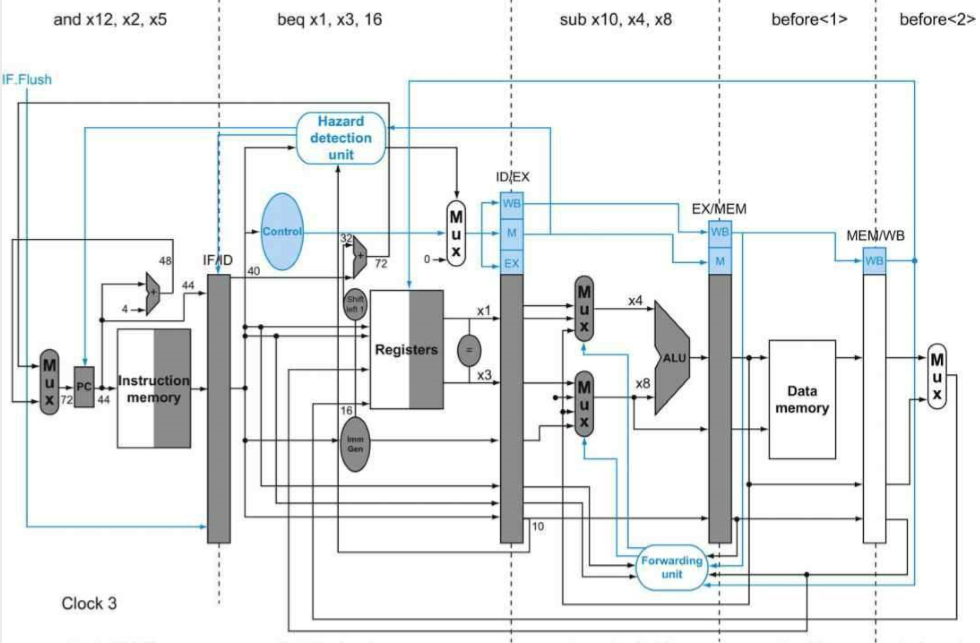


* Os estágios ID e EX foram simplificados, faltam a unidade de extensão de sinal e a lógica do branch nos estágios, respectivamente.
* As bolhas reduzem o desempenho, mas são necessárias para obter resultados corretos. O compilador pode organizar o código para evitar conflitos e necessidade de bolhas.

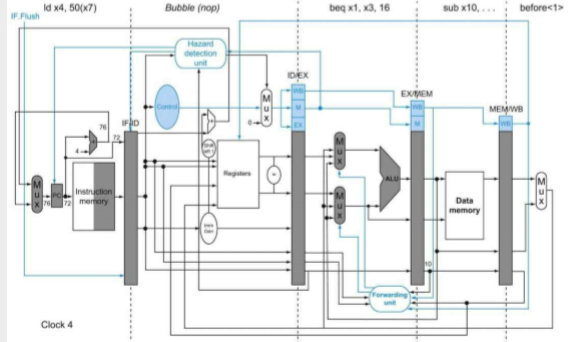
**4.8 Control Hazards**

* Uma instrução deve ser buscada em cada ciclo de clock para manter o pipeline, porém, neste design, a decisão do desvio só é feita no estágio MEM do pipeline.
* Como o branch só decide sobre o desvio no quarto ciclo de clock, no estágio MEM, 3 instruções após o branch serão buscadas e terão sua execução iniciada. Se não houver intervenção, essas 3 instruções seguintes começaram a executar antes do branch tomar sua decisão.
* Control hazards ocorrem bem menos frequentemente do que data hazards.
* Previsão: supor que o desvio não deve ocorrer e continuar a execução com as instruções seguintes; se o desvio ocorrer, as instruções que foram buscadas e começaram a ser executadas devem ser descartadas; a execução deve continuar agora com a instrução correta indicada pelo branch; se os desvios, na metade do tempo, não ocorrerem e se o custo de descartar instruções for pequeno, essa estratégia resolve os control hazards.
  + Para descartar instruções basta mudar o valor dos sinais de controle para 0, como foi feito para criar uma bolha. A diferença é que devemos também mudar as três instruções nos estágios IF, ID e EX quando o branch chegar no estágio MEM.
  + Descartar instruções é chamado de flush.
* Reduzir o delay dos branches: fazer com que o branch execute mais cedo no pipeline, pois, dessa forma, menos instruções precisarão ser descartadas.
  + Para adiantar a execução do branch são necessárias duas coisas: fazer o cálculo do endereço de desvio e avaliar a decisão de desvio.
  + Facilmente podemos adiantar o cálculo do endereço de desvio, pois já temos o PC e o imediato no registrador IF/ID, basta mover o somador do branch do estágio EX para o estágio ID.
  + O cálculo de endereço de desvio será calculado para todas as instruções, porém só será usado quando necessário.
  + *Para o BEQ:* precisamos comparar os dados de dois registradores no estágio ID para ver se são iguais; a igualdade pode ser testada fazendo XOR’s bit a bit, e realizando uma OR com os resultados dos XOR’s; mover o teste de desvio para o estágio ID implica na adição extra de hardware de forwarding e de detecção de conflitos, pois se um branch dependente do resultado de uma instrução que ainda não foi finalizada também deve funcionar corretamente; pode ser necessário adiantar a informação para a unidade que testa a igualdade durante o estágio ID do branch.
  + Durante o estágio ID, devemos decodificar a instrução, decidir se um forwarding será ou não necessário e testar a igualdade dos registradores, para que se for um branch e forem iguais, carregarmos o PC com o endereço de desvio. A introdução da unidade de teste de igualdade no estágio ID faz necessária uma nova lógica de forwarding. Os dados de um branch que talvez precisem ser adiantados podem vir do registrador EX/MEM ou do MEM/WB.
  + Como o valor para a comparação de um branch é necessário no estágio ID, mas pode ser produzido apenas em um ciclo de clock posterior, é possível a ocorrência de um data hazard fazendo necessária o uso de uma bolha.
  + Se uma instrução da ULA precede imediatamente um branch que depende do resultado da ULA, uma bolha será necessária, pois o estágio EX para a instrução da ULA ocorrerá depois do estágio ID do branch.
  + Se uma instrução de load precede imediatamente um branch que depende do valor lido pelo load, duas bolhas serão necessárias, pois o resultado do load aparece apenas no final do estágio MEM do load, porém ele é necessário no início do estágio ID do branch.
  + Mesmo com essas dificuldades, mover a execução do branch para o estágio ID é um grande avanço, pois reduz a penalidade de um branch para apenas uma única instrução caso o desvio ocorra.
  + Para dar um flush na instrução que se encontra no estágio IF, adicionamos um sinal chamado IF.Flush, que zera a instrução guardada no registrador IF/ID. Isso transforma a instrução que seria executada em uma nop.

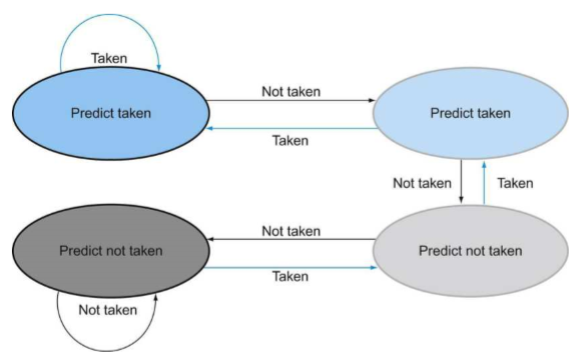
**Antes da realização do desvio:**



**Depois da realização do desvio:**

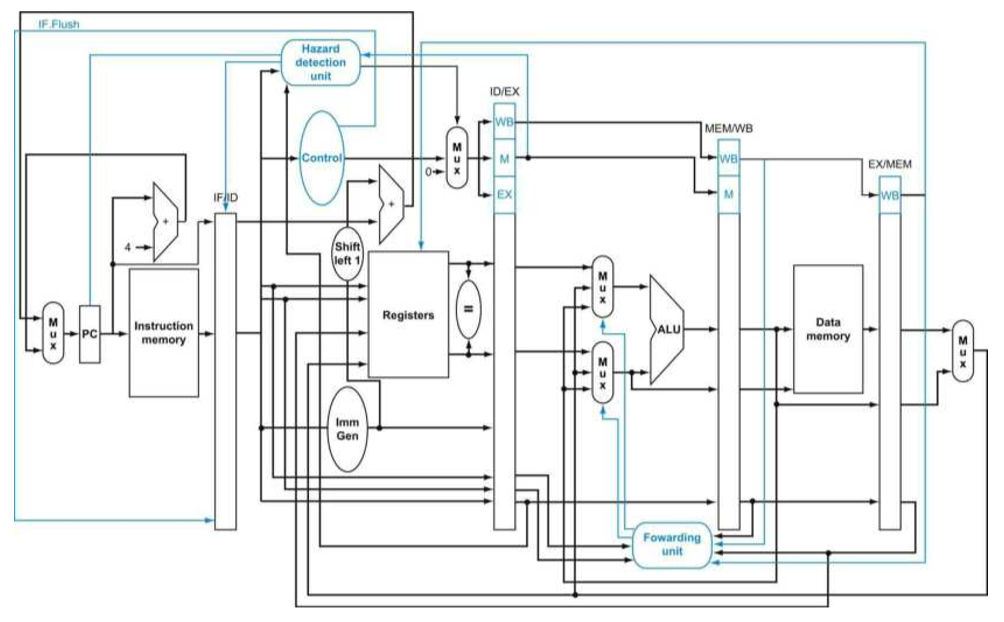


* Dynamic Branch Prediction: com mais hardware é possível tentar prever o comportamento do desvio durante a execução do programa.
  + Procurar o endereço da instrução para ver se o desvio ocorreu ou não na última vez que esse branch foi executado, e se tiver ocorrido, buscar a próxima instrução do mesmo lugar que buscou da última vez.
  + Branch prediction buffer/branch history table: uma pequena memória que é indexada com a parte mais à direita do endereço da instrução do branch e que contém um ou mais bits indicando se o desvio foi executado ou não recentemente.
  + Nós não sabemos com certeza se a previsão é a correta (talvez tenha sido colocada lá por outra instrução de branch que possui os bits mais à direita iguais a esta), porém isso não afeta a precisão. A previsão é somente um palpite que torcemos para que esteja correto. Buscamos a instrução prevista, se a previsão tiver sido incorreta, descartamos a instrução buscada, invertemos o bit da tabela de histórico de branches e buscamos a instrução correta a ser executada.
  + Exemplo: um desvio de loop em que o desvio ocorre 9 vezes seguidas e na 10ª vez não ocorre.
    - A previsão será incorreta no primeiro e no último loop. Errar a previsão do último loop é inevitável, já que o bit de previsão indicará a ocorrência do desvio (pois ocorreram 9 desvios antes). O erro da primeira previsão ocorre, pois no final da execução do loop anterior, o desvio não ocorreu, logo isso foi indicado na tabela. Logo, a precisão das previsões para esse branch seria 80%.
    - O objetivo é que a previsão de branches regulares como esse seja igual a frequência de desvios ocorridos (nesse caso deveria ser 90%). Para resolver esse problema, usamos previsões de 2 bits, que consiste no fato de que uma previsão deve errar 2 vezes antes de ser mudada.



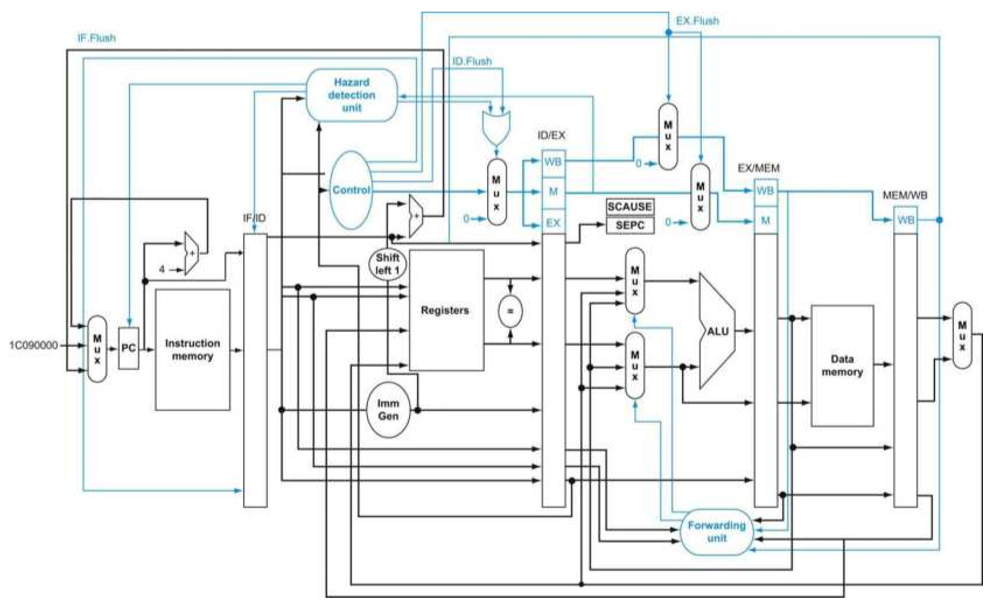
* O branch prediction buffer é implementado como um pequeno buffer que é acessado com a instrução no estágio IF.
* Se o desvio é prevista como ocorrido, a busca de instrução começa exatamente quando o endereço de desvio estiver pronto.
* Se for previsto como não ocorrido, a próxima instrução (PC+4) é buscada e executada sequencialmente.
* Se a previsão acabou errando, o bit é mudado na tabela.
* A previsão só nos diz se o desvio tende a ocorrer ou não, ainda sim é necessário calcular o endereço de desvio. Para facilitar, é usado um branch target buffer, uma cache que guarda o valor do PC após os branches em execuções passadas.
* Correlating predictor: faz a previsão de branches relacionando comportamento local (desse branch em específico) e comportamento global (de todos os branches do código). Isso aumenta a precisão das previsões sem aumentar o número de bits da tabela.
* Tournament branch predictor: usa várias unidades de previsão e escolhe a previsão que tem provido os melhores resultados. O mais comum usa a previsão baseada em dados locais e a previsão baseada em dados globais, escolhendo a que tem acertado mais.

**Caminho de dados final para Control Hazards:**



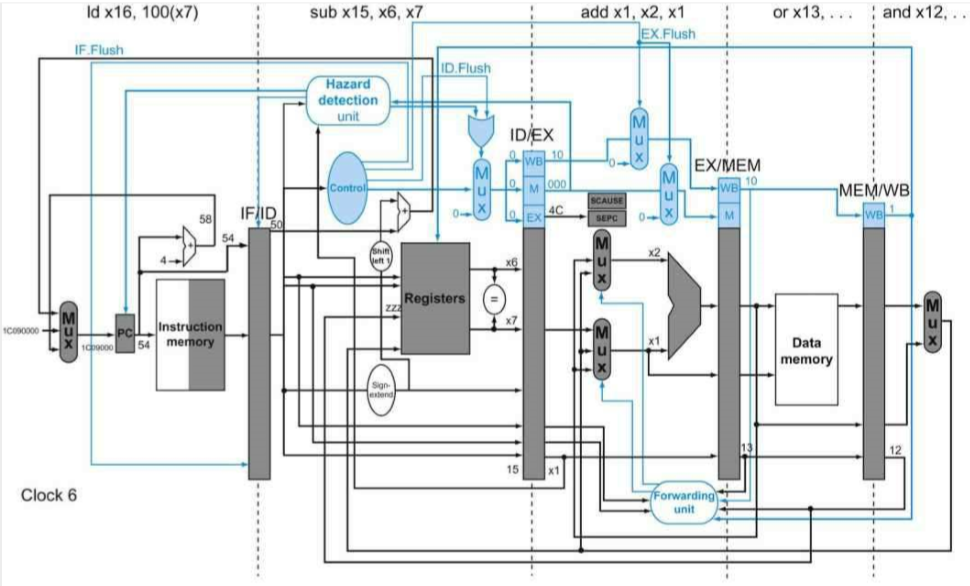
**4.9 Exceções**

* Uma das responsabilidades da unidade de controle do pipeline é ligar com exceções e interrupções. Ex: instrução desconhecida.
* Exceção: também chamada de interrupção; um evento não esperado que interrompe a execução do programa; opcode inexistente, overflow, divisão por zero, etc...
* Interrupção: uma exceção que vem de fora do processador; algumas arquiteturas usam o termo interrupção para todas as exceções; dispositivos de entrada/saída.
* Usamos exceção para qualquer evento inesperado sem distinguir se a causa é interna ou externa. Usamos interrupção apenas quando evento é causado externamente.
* Mau funcionamento do hardware: a ação básica que o processador deve tomar é armazenar o endereço da instrução que causou o problema no registrador SEPC e transferir o controle para o sistema operacional em um endereço específico.
* O sistema operacional, após tratar a exceção, pode continuar a execução do programa ou não. Caso deseje continuar, usa o registrador SEPC para determinar onde deve recomeçar a execução.
* Para o sistema operacional lidar com a exceção ele deve saber qual foi a sua causa. Para isso, no RISC-V, existe um registrador SCAUSE que contém um indicador para qual foi a causa da exceção.
* Ou seja, para realizar o tratamento de exceções precisamos apenas de alguns registradores e sinais de controle extras:
  + SPEC: registrador de 64 bits que guarda o endereço da instrução causadora de exceção.
  + SCAUSE: registrador de 64 bits (apenas poucos bits são realmente usados) que guarda a causa da exceção.
* A implementação do pipeline trata exceções como se fossem um control hazard.
* Vamos supor que existe um mau funcionamento do hardware na instrução add. Devemos realizar um flush nas instruções seguintes ao add e começar a buscar instruções a partir do novo endereço (rotina de tratamento).
* Para dar um flush na instrução no estado ID, usamos o mux pertencente a este estágio que zera os sinais de controle para bolhas. Um novo sinal e controle chamado ID.Flush é colocado numa OR juntamente com o sinal de bolha da unidade de detecção de conflitos, o resultado dessa OR é quem determina se o flush ocorre ou não.
* Para dar um flush na instrução do estado EX, usamos um novo sinal chamado EX.Flush para fazer com que novos mux’s coloquem zero nos sinais de controle.
* Para escrever no PC o endereço da rotina de tratamento, basta adicionar mais um input no mux do PC.
* As mudanças no circuito abaixo são:
  + Novo input ao mux do PC.
  + O registrador SCAUSE e o registrador SEPC.

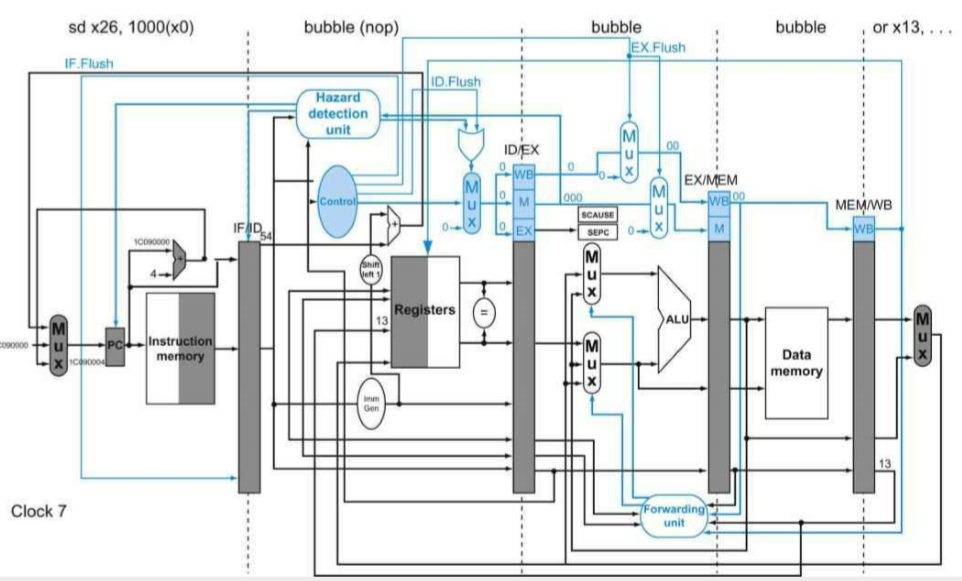


* Algumas exceções necessitam que, eventualmente, a instrução que a causou seja executada. A maneira mais fácil de fazer isso é dar um flush na instrução e recomeçar a execução da instrução completamente depois da exceção ter sido tratada.
* Comportamento do pipeline:
  + Instruções antes da que causou exceção devem ser terminadas.
  + Instruções após a que causou exceção devem ser anuladas.
* Dada a seguinte sequência e um mal funcionamento do hardware na instrução de add:
  + sub x11, x2, x4
  + and x12, x2, x5
  + or x13, x2, x6
  + add x1, x2, x1
  + sub x15, x6, x7
  + ld x16, 100(x7)
* Em caso de exceções a seguinte rotina é executada:
  + sd x26, 1000(x10)
  + sd x27, 1008(x10)

**Antes da detecção da exceção:**



**Depois da detecção da exceção:**



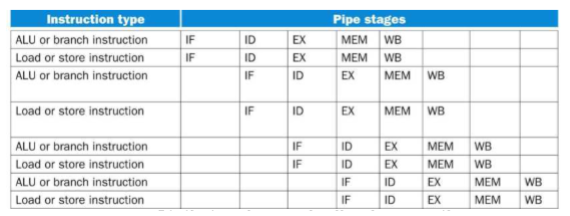
* A exceção é detectada no estágio EX, no clock 6. O endereço da instrução de add é salva no SEPC. Isso faz com que ocorra um flush em todos os sinais que seriam determinados ao final desse estágio. Zerando os sinais de controle da instrução de add. Além disso, as instruções que começaram a execução depois do add também são descartadas, substituídas por bolhas. Além disso, é buscada a primeira instrução da rotina de tratamento.
* Note que as instruções and e or, que foram iniciadas antes do add, têm sua execução terminada.
* Com cinco instruções ativas no mesmo ciclo de clock, é um desafio associar a exceção com a instrução correta. Além disso, múltiplas exceções podem ocorrer no mesmo ciclo de clock. Para resolver esse problema, priorizamos as exceções para determinar mais facilmente qual será tratada primeiro.
* Pedidos de dispositivos de E/S e mau funcionamento do hardware não são associadas com uma instrução específica.
* Hardware: deve parar a instrução que causou a exceção no meio de sua execução, dar um flush nas instruções seguintes, deixar as instruções anteriores terminarem de executar, guardar a causa da exceção, guardar o endereço da instrução que causou a exceção, e desviar para o endereço correto.
* Software: avaliar a causa da exceção e agir apropriadamente.
* Interrupção/Exceção precisa: o endereço correto é salvo em SEPC.
* Interrupção/Exceção imprecisa: o valor corrente do PC (não da instrução que causou a exceção) é salvo em SEPC.

**4.10 Paralelismo por Instruções:**

* Filosofia RISC (Reduced Instruction Set Computer): poucas instruções, simples e com formato único; modos simples de endereçamento; implementação mais eficiente.
* Pipeline explora o potencial paralelismo entre instruções. Esse paralelismo é chamado de instruction-level parallelism (ILP).
* Dois métodos principais para aumentar o paralelismo: aumentar o número de estágios (profundidade do pipeline), superpipeline, para serem comportadas mais instruções num mesmo datapath, menos trabalho por estágio, menor período de clock; ou replicar componentes internos do processador para que múltiplas instruções possam atuar em um único estado (multiple issue/despacho múltiplo), superescalar.
* Existem duas maneiras de implementar um processador de despacho múltiplo, a maior diferença entre elas é a divisão de trabalho entre o compilador e o hardware. Essa divisão determina se as decisões estão sendo feitas estaticamente (durante a compilação) ou dinamicamente (durante a execução).
* Static multiple issue: estratégia de implementar um processador multiple-issue onde muitas decisões são tomadas antes da execução pelo compilador.
* Dynamic multiple issue: estratégia de implementar um processador multiple-issue onde muitas decisões são tomadas durante a execução.
* Pontos importantes a serem tratados:
  + Issue slots: conjunto de instruções; como ocorre a divisão das instruções para a execução; para os estáticos, esse processo é feito pelo compilador; para os dinâmicos, esse processo é feito durante a execução pelo processador, apesar de que, o compilador normalmente vai tentar ajudar reorganizando as instruções em uma ordem mais benéfica.
  + Data e control hazards: nos estáticos, o compilador lida com algumas ou todas as consequências de conflitos estaticamente; já nos dinâmicos, os conflitos são resolvidos durante a execução usando técnicas de hardware.
* Especulação: abordagem que permite ao compilador ou ao processador “adivinhar” sobre as propriedades de uma instrução; “adivinhar” o resultado de uma instrução para remover dependências entre outras instruções.
  + Podemos tentar prever o resultado de um desvio, para que as instruções após o branch já possam ser executadas.
  + Podemos tentar especular que um store que procede um load não faz referência ao mesmo endereço, o que permite que façamos o load primeiro antes do store.
  + Porém, as especulações podem ser incorretas.
  + Por isso, qualquer especulação deve possuir um mecanismo para checar se o palpite foi correto e um método para desfazer os efeitos das instruções que foram executadas na especulação.
  + O compilador pode reordenar instruções.
  + O processador pode fazer o mesmo, porém durante a execução usando técnicas de hardware.
  + Em caso de **especulação errada**:
    - Pelo software: normalmente são inseridas instruções que checam a precisão da especulação e que provem uma rotina de tratamento que deve ser usada caso a especulação tenha sido incorreta.
    - Pelo hardware: o processador normalmente guarda em buffers os resultados das especulações até que ele saiba com certeza que não são apenas especulações; se a especulação foi correta, as instruções são finalizadas permitindo que o conteúdo dos buffers sejam escritos nos registradores ou na memória; se a especulação foi incorreta, o hardware faz um flush nos buffers e re-executa a sequência correta de instruções.
    - Normalmente, quando a especulação é errada, o pipeline precisar receber um flush ou pelo menos uma bolha, diminuindo assim a performance.
* Um outro problema possível da especulação é que fazendo isso com determinadas instruções pode introduzir exceções que não estavam anteriormente presentes:
  + Por exemplo, suponha uma instrução de load movida de forma especulativa. Mas o endereço que ele usa não está dentro dos limites quando a especulação está incorreta. O resultado seria que uma exceção que não deveria ter ocorrido, mas ocorreu.
  + Na especulação baseada em compilador, esses problemas são evitados pela adição de suporte especial de especulação, que permite que tais exceções sejam ignoradas (não tratadas) até que fique claro que eles realmente devem ocorrer.
  + Na especulação baseada em hardware, as exceções são simplesmente armazenadas em buffer até que seja claro que a instrução que os causa não é mais especulativa e está pronta para ser concluída. Nesse ponto, a exceção é gerada e o tratamento de exceção continua normalmente.

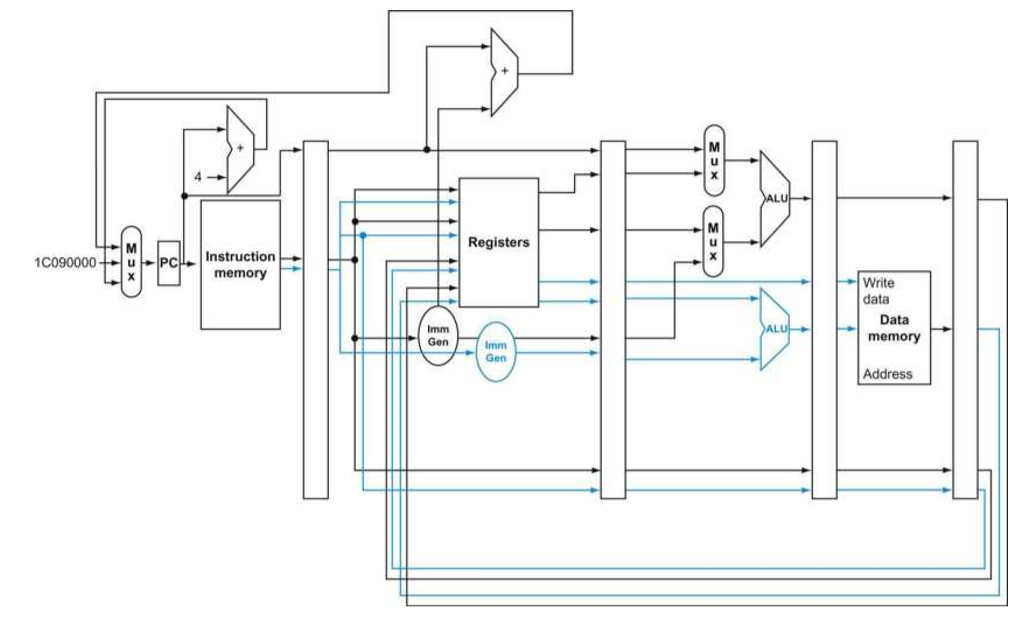
**Static Multiple Issue:**

* Como temos o auxílio do compilador para ajudar com o empacotamento das instruções emitidas num ciclo de clock, o mesmo agrupa isso em issue packets, conhecidas como Very Long Instruction Word (VLIW); nesse tipo de escalonamento, o compilador deve remover todos os conflitos.
  + Issue packet**:** o conjunto de instruções que são gerados em conjunto em um ciclo de clock; pode ser determinado estaticamente pelo compilador ou dinamicamente pelo processador, que funciona como uma instrução grande (VLIW) com várias operações, e traz múltiplas instruções concorrentes; essas instruções são determinadas pelos recursos do pipeline.
  + Very Long Instruction Word (VLIW): um estilo de instrução do conjunto de arquitetura que inicia muitas operações que são definidas para serem independentes umas das outras, em uma instrução de largura única, normalmente com muitos campos de opcode separados.
  + As responsabilidades do compilador podem incluir previsão de ramificação estática e agendamento de código para reduzir ou evitar todos os perigos.
* Exemplo RISC-V ISA: uma das instruções pode ser uma operação na ULA ou um branch e a outra instrução pode ser um load ou um store; buscando duas instruções por ciclo faz com que sejam lidos 64 bits de instruções; como as instruções estão sempre em pares, se não é possível parear uma instrução, colocamos um nop junto com ela; dessa forma as instruções estão sempre em pares, sendo a outra possivelmente um nop.
* É importante observar que a operação da ULA e a instrução de acesso à memória são iniciadas ao mesmo tempo.



* O tratamento do compilador varia de acordo com o tipo do processador em relação a forma que eles lidam com potential data e control hazards.
  + Em alguns designs, o compilador assume total responsabilidade pela remoção de todos os perigos, agendamento do código e inserção de nop’s para que o código seja executado sem qualquer necessidade de detecção de perigos ou stalls geradas por hardware.
  + Em outros, o hardware detecta esses perigos, gera uma bolha entre dois issue packets enquanto exige que o compilador evite todas as dependências dentro de um pacote de instrução.
* Data Hazard:
  + add x10, x0, x1
  + ld x2, 0(x10)
  + Precisamos dividir essas instruções em pacotes diferentes, porém isso causa um retardo.
  + Resultados de operações de ULA não podem ser usados pela instrução que é pareada com ela.
* Load-Use Hazard: o resultado de um load não pode ser usado no clock seguinte, mas isso significa que não pode ser usado pelas duas instruções seguintes; temos que um escalonamento mais agressivo é necessário.

**Mostrando a duplicação de hardware:**



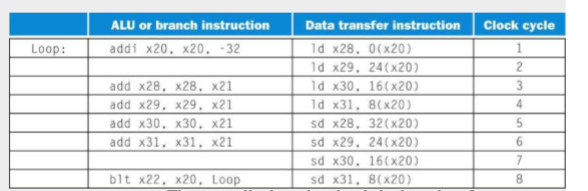
* O processador mostrado acima pode aumentar a performance em até um fator de 2. Porém, para isso, ele necessita que o dobro de instruções sejam colocadas em execução e isso aumenta a perda de performance em relação a conflitos de dados e de controle.
* Use latency: número de ciclos de clock entre uma instrução de load e uma instrução que pode usar o resultado do load sem causar um stalling no pipeline.
* Loop:
  + ld x31, 0(x20)
  + add x31, x31, x21
  + sd x31, 0(x20)
  + addi x20, x20, -8
  + blt x22, x20, Loop
* As três primeiras instruções possuem dependência, assim como as duas últimas também.

**Solução:**



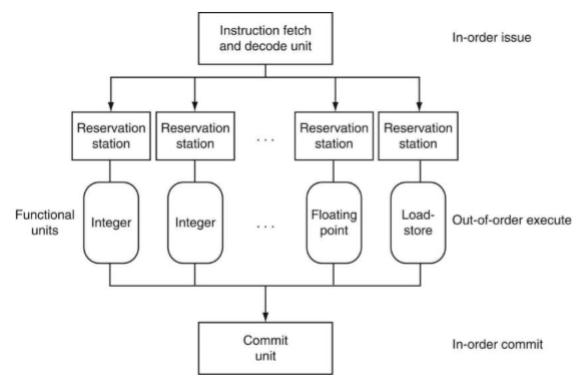
* Uma técnica do compilador importante para obter mais desempenho dos loops que acessam arrays é o Loop Unrolling, no qual várias cópias do corpo do loop são feitas e instruções de diferentes interações são programadas (agendadas) juntas.
  + Register renaming/Renomeação de Registradores: adição de registradores extras; eliminar dependências que não são verdadeiramente de dados, mas que poderiam levar a conflitos; conhecido como antidependência ou dependência de nome, acontece quando o mesmo registrador é usado repetidamente, porém os dados manipulados não são os mesmos.
  + Renomeando os registradores durante o unrolling permite que o compilador mova essa instruções independentes para melhor combinar o código. O processo de renomeação elimina dependências de nome, deixando apenas dependências de dados.
  + O custo desse aumento de performance é o uso de mais registradores temporários e o aumento no tamanho do código.

**Usando o loop unrolling para o exemplo anterior:**



**Dynamic Multiple Issue:**

* Também são conhecidos como processadores superescalares; as instruções são iniciadas em ordem e o processador decide se são zero, uma ou mais instruções a serem executadas em um determinado ciclo de clock; evitam conflitos de dados e estrutural.
  + Ainda assim, para atingir um bom desempenho, o compilador continua tentando organizar as instruções para afastar as dependências.
  + O código, seja ele organizado ou não, é garantido pelo hardware que irá executar corretamente, pois não é responsabilidade do compilador.
  + Evitaa necessidade de escalonamento pelo compilador, apesar das otimizações serem bem vindas.
  + Permite a CPU executar as operações fora de ordem, para evitar stalls (mas mesmo assim, ela atualiza os registradores na ordem).
  + Exemplo:
    - ld x31, 0(x21)
    - add x1, x31, x2
    - sub x23, x23, x3
    - andi x5, x23, 20
  + Mesmo com a sub pronta para executar, precisamos esperar o load e o add terminarem.
  + Temos um despacho em ordem, uma execução fora de ordem e uma escrita em ordem ou fora de ordem.
* Dynamic Pipeline Scheduling: escolhe quais instruções devem executar em um determinado ciclo de clock enquanto tenta evitar conflitos e bolhas; suporte de hardware.
  + O pipeline é dividido em três unidades:
    - Busca e Emissão de instrução: busca instruções, decodifica elas e as envia para a unidade funcional correspondente.
    - Unidades funcionais: todas possuem buffers chamados reservation stations/estações reservas, que guardam os operandos e a operação; assim que o buffer contém todos os operandos e a unidade funcional está pronta para executar, o resultado é calculado; depois disso ele é enviado para alguma estação reserva que estava esperando por esse dado, além disso armazena-o na commit unit.
    - Commit unit: guarda os resultados das operações até que seja seguro armazená-los num registrador ou na memória; o seu buffer é chamado de reorder buffer/buffer de reordenação; é a unidade em um pipeline de execução dinâmica que decide quando é seguro liberar o resultado de uma operação para registradores ou para a memória.



* A combinação do armazenamento dos operandos nas estações de reserva/reservation station e do resultado no reorder buffer/buffer de reordenação fornece uma forma de renomeação de registrador.
  + Quando uma instrução começa a ser processada, os operandos são salvos na estação de reserva/reservation station apropriada. Após isso, se ocorrer escrita sobre o registrador cópia dos operandos, não ocorrerá problemas, pois eles já foram armazenados.
  + Se um operando não está em um register file e nem em um reorder buffer/buffer de reordenação, deve está esperando para ser produzido por uma unidade funcional. O nome dessa unidade funcional é rastreado e quando ela eventualmente produz o dado, ele é copiado diretamente na reservation station.
  + Ou seja, reorder buffer e reservation stations são usadas para realizar o register renaming.
* Out of order execution: instruções podem ser executadas em uma ordem diferente da qual elas foram buscadas; uma instrução bloqueada de executar não faz as instruções seguintes esperar.
* In order commit: o resultado de uma operação são escritos em locais visíveis ao programador (registradores e memória) na mesma ordem em que as instruções são buscadas.
* Para fazer o programa funcionar como se ele estivesse rodando in-order pipeline, a unidade de busca de instrução e decodificação deve iniciar as instruções na ordem, o que permite que as dependências sejam rastreadas e a commit unit deve escrever os resultados nos registradores e na memória na ordem em que as instruções são buscadas.
* Portanto, se ocorrer uma exceção, o computador pode apontar para a última instrução executada e serão atualizados apenas os registros escritos por instruções antes da instrução que causou a exceção.
* Embora o front-end (busca e emissão) e o back-end (commit) do pipeline sejam executados em ordem, as unidades funcionais são livres para iniciar a execução sempre que os dados necessários estiverem disponíveis.
* O dynamic scheduled pipeline também pode oferecer suporte à especulação sobre endereços de load, permitindo a reordenação de load/store e o uso da commit unit para evitar especulações incorretas.
* Há 3 razões para o uso do dynamic scheduling (agendamento dinâmico):
  + Nem todas as bolhas são previsíveis. Em particular, erros de cache causam bolhas imprevisíveis. A programação dinâmica permite que o processador esconda algumas daquelas bolha, continuando a executar instruções enquanto espera a bolha terminar.
  + Se o processador especular os resultados dos branches usando dynamic branch prediction (previsão dinâmica de ramificação), ele não poderá saber a exata ordem das instruções em tempo de compilação, pois depende no comportamento previsto e real dos branches.
  + Como a latência do pipeline e a largura do problema mudam de uma implementação para outra, a melhor maneira de compilar uma sequência de código também é alterada. O dynamic scheduled permite que o hardware oculte a maioria desses detalhes.
* Mesmo existindo processadores com 4 à 6 instruções por clock, pouquíssimas aplicações sustentam mais do que duas instruções por clock. Isso ocorre por dois motivos:
  + A maioria das dependências que diminuem a performance, são conflitos que não podem ser resolvidos com total eficácia, reduzindo o paralelismo entre as instruções. Muitas vezes o compilador ou hardware não sabem exatamente se uma dependência existe ou não, e assim devem assumir conservativamente a dependência existe.
  + As perdas na hierarquia de memória também limitam a capacidade de manter o pipeline cheio.

**Eficiência Energética:**

* O lado ruim de aumentar a exploração de paralelismo de instruções por meio de dynamic multiple issue e especulação é a potencial ineficiência energética.
* Mesmo processadores mais simples não sendo tão rápidos quantos os sofisticados, eles podem ter uma melhor performance por joule.
* Temos um problema de energia vs número de transistores.

**Dependência de Dados:**

* Temos a dependência verdadeira, dada por read-after-write (RAW).
* Temos a dependência de saída, dada por write-after-write (WAW).
* Temos a antidependência, dada por write-after-read (WAR).
* Para tratá-las:
  + Inserção de nops/bolhas.
  + Inserir instruções que não dependem das que causam a dependência (reorganização de código).
  + Usar armazenamento temporário.